National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Services des thèses canadiennes

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL 339 (r 86/01)

Canada

71

0-315-23269-2

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Division
Division des thèses canadiennes

Ottawa, Canada
K1A 0N4

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

AVINASH VASHISTHA

Date of Birth — Date de naissance	Country of Birth — Lieu de naissance
23 April 1961	India

Permanent Address — Résidence fixe

C-84, Nirala Nagar
Lucknow, U.P.
INDIA - 226007

Title of Thesis — Titre de la thèse

Analysis of Overheads in a Multiprocessor System.

University — Université

University of Alberta

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

Master of Science

Year this degree conferred — Année d'obtention de ce grade	Name of Supervisor — Nom du directeur de thèse
1985	Dr. Tony A. Marsland

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date	Signature
2 August 1985	Avinash Vashistha

NL-91 (4/77)

The University of Alberta

Analysis of Overheads in a Multi-processor System

by

© Avinash Vashistha

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1985

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Avinash Vashistha

TITLE OF THESIS: Analysis of Overheads in a Multi-processor System

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1985

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) Avinash Vashistha

Permanent Address:
C-84, Nirala Nagar
Lucknow, U.P.
India - 226007

Dated 1 August 1985

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Analysis of Overheads in a Multi-processor System** submitted by **Avinash Vashistha** in partial fulfillment of the requirements for the degree of **Master of Science.**

T. A. Marsland
.....
Supervisor

.....

.....

.....

Date 1 August 1985

ABSTRACT

The evolution of multiprocessor systems as a means for faster and inexpensive computation has triggered a need for the design of efficient parallel algorithms. Parallel algorithms for a multiprocessor system present some unique aspects in both their design and their analysis. This thesis deals with the design issues affecting a parallel implementation of the alpha-beta algorithm used for game tree search. A modified version of parallel chess program *Parabelle* has been implemented on a network of SUN workstations. An analytical model of a parallel implementation of the alpha-beta algorithm on a multiprocessor system is discussed and various search constants associated with it are experimentally determined.

When dealing with parallel algorithms, in particular the parallel implementation of the alpha-beta algorithm, it is not possible to get an 'N' fold speedup using 'N' processors. This behavior has been attributed to the overheads involved, principally search, communication and synchronization overheads. These types of overheads are analyzed. The experimental results presented in this thesis confirm that synchronization overhead accounts for most of the degradation in processor utilization. A scheme has been proposed to reduce the synchronization loss.

Acknowledgements

I wish to express my gratitude to my supervisor Dr. Tony Marsland for his guidance, assistance and encouragement throughout the course of this work.

I am also grateful to the members of my examining committee, Dr. W. Joerg , Dr. J. Tartar and Dr. Lee White, for their helpful comments.

I am indebted to the members of my family back home and to my wife Garima for their continual love, encouragement and support.

Table of Contents

Chapter	Page
Chapter 1: INTRODUCTION	1
1.1. Why Multiprocessor Systems and Parallel Processing?	1
1.2. Development of Multiprocessor Systems and Parallel Algorithms	2
1.3. Problems in Design of Parallel Algorithms	4
1.4. Overview of the Thesis	5
Chapter 2: MULTIPROCESSOR SYSTEMS AND PARALLEL TREE SEARCH ALGORITHMS	7
2.1. Multiprocessor Architectures	7
2.1.1. C.mmp and Cm*	7
2.1.2. ZMOB : A Mob of 256 Z-80 Processors	8
2.1.3. Distributed Multi-processor Systems	9
2.1.4. Processor Tree	10
2.2. Sequential Tree Search Algorithms	10
2.3. Parallel Tree Search Algorithms	12
2.4. The Parallel Minimal Window Search	14
Chapter 3: SOFTWARE AND HARDWARE SYSTEM CONFIGURATION	17
3.1. Virtual Tree Machine Implementation	18
3.2. Inter-unit Communication	22
3.3. Inter-process Communication	23
Chapter 4: PERFORMANCE ISSUES	25
4.1. Overheads	25
4.1.1. Communication Overhead	28
4.1.2. Search Overhead	29
4.1.3. Synchronization Overhead	30
4.2. Analytical Model	31
4.3. Analysis of Experimental Results for the Analytical Model	32
4.4. Results for a 5-ply Search	35
Chapter 5: SYNCHRONIZATION OVERHEAD	41
5.1. Solutions to the Synchronization Problem	42
5.2. Scheme to Reduce Synchronization Overhead	43
5.3. Experimental Results	58
5.3.1. Processor Idle Time	59

5.3.2. Processor Re-scheduling Overhead 62
5.4. Conclusion on the Effectiveness of the Proposed Scheme 63
Chapter 6: CONCLUSIONS 65
6.1. Directions for Future Work 66
References 68
Supplementary Bibliography 71

List of Tables

Table		Page
4.1	Node counts for a 5-ply Search / Single processor	33
4.2	Multiprocessor Results from a 5-ply Search	36
5.1	Processor Idle Times for a 4 Processor System	61

List of Figures

Figure		Page
3.1	A Sample Processor Tree with SUN Processors	19
3.2	Processor Tree Hardware Configuration	21
4.1	Working and Idle Time of Processors	27
4.2	Speedup vs No. of Processors	37
4.3	Average Overheads for a 5-Ply Search	38
5.1	Processor Tree for Current Implementation	45
5.2	Configuration File for Current Implementation	45
5.3	Processor Tree for Proposed Implementation	47
5.4	Configuration File for Proposed Implementation	47
5.5	PMWS Algorithm for Root Processor	48
5.6	PMWS Algorithm for Slave Processors	49
5.7	Interrupt Handler Routine for Root Processor	50
5.8	Algorithm for Finding the Assigned Master.	52
5.9	Tree Search After Re-scheduling of Processors	53
5.10	Interrupt Handler Routine for Assigned Master	55
5.11	Search Algorithm for Assigned Master	56
5.12	Search Algorithm for Slave Processor	57
5.13	Processor Idle Time Measurement	60

Chapter 1

INTRODUCTION

1.1. Why Multiprocessor Systems and Parallel Processing?

The quest for higher speed, higher performance and more reliable computers at lower costs goes on relentlessly. This is accelerating as the technologically advanced societies rely more and more on computers in nearly all aspects of life. Technology has succeeded in providing us with faster and more powerful computers. Over the past few years much attention has been given to microprocessors, their costs have drastically decreased and we now have inexpensive processors that provide a high speed of operation and powerful capabilities. Many such units can collectively be used to do a task faster than a single machine. This has resulted in an increased interest in **Multi-microprocessor Systems.**

A large amount of research has gone into designing efficient multi-processor systems. Most attention has been given to 2-6 processor systems, however it is conceivable that many multiprocessor systems composed of 1000 or more processors will be built soon [MYRI84]. This evolution has brought about the need for the development of efficient **parallel algorithms.** Analogous to a management environment, a multiprocessor system has problems of scheduling work, distributing control and efficiently communicating between processors. This thesis attempts to highlight the issues involved in the design of efficient parallel algorithms for a multi-processor system.

1.2. Development of Multiprocessor Systems and Parallel Algorithms

Ever since digital computers were first introduced in the early fifties, there has been a constant thrust to achieve more computing power. Architectural ingenuity and improved technology have resulted in more powerful and faster computers over the years. During a given period when performance limitations of a current technology are reached, duplication of hardware remains a tempting option to go beyond the capability of a single machine. This is also a means of improving machine availability and reliability.

With the availability of low cost computers, people tend to split complex computational tasks into individual components and to allocate separate identical processors to the constituent parts. The structure of many real-time processes lend themselves to such an approach. The developments that are of interest in this thesis work are in the multiprocessor environment, where several identical digital computers are linked together to operate in parallel while communicating and cooperating with each other to achieve a common objective of searching a tree.

The idea of using a multiplicity of computers for improved reliability was applied in the early stages of computer development, e.g. PILOT, in 1958. "Polymorphic" computer is another example where several computer modules, buffers and peripherals can be interconnected through a switching matrix to increase machine availability as well as improving processing power owing to parallel processing and sharing of peripherals. However, the way multiprocessing is achieved varies from one machine to the other. For a general overview, Baer [BAER76] offers an introductory presentation of multi-processor architectures; Jones [JONE80] presents a report on his experiences with multi-processor systems; and Enslow [ENSL77] surveys some specific multiprocessor organizations.

Flynn [FLYN66] classifies the machines based on the instruction and data stream architecture. Consistent with Flynn's classification, array processors are typical examples of SIMD (Single Instruction Multiple Data stream) machines. Array processors were developed for special purpose applications that required sequential execution of the same instruction stream (mainly numerical operations) on parallel streams of data. ILLIAC IV [KUCK68] is an example of such a machine.

Another class of multi-processors are the MIMD (Multiple Instruction Multiple Data stream) machines. C.mmp and Cm*, built at Carnegie-Melon University, are examples of this class. The C.mmp machine has 16 processors that share 16 memory modules over a cross-bar switch [WULF72]. Cm* is a multi-microprocessor system based on DEC LSI-11, where the interconnection of computer modules is achieved by a hierarchy of parallel busses [SWAN77].

With the arrival of microcomputers, the interest in building large multi-computer structures has grown considerably over the last few years. Myrias 4000, a large multi-processor system being built by Myrias Research, is a supercomputer built from more than 4000 processing units [MYRI84]. The architecture of Myrias 4000 is particularly suitable for linear algebra applications. Another machine, ZMOB [RIEG79][BANE81], is a multiprocessor system being built from a collection of 256 identical but autonomous Z-80A based processors.

Many practical problems can be solved using tree structured algorithms. It appears that such problems can be efficiently solved by using many processors connected in form of a tree [BURT84]. An effective solution which is an alternative to tree machines is the **Virtual Tree Machine** (VTM) [BURT84]. The key components of a virtual tree machine are a network of interconnected processors and an algorithm to decide when and where to run each process. In this thesis, a virtual tree machine has been used to study the issues involved in multi-processing

and the design of parallel algorithms for such a multi-processor system.

1.3. Problems in Design of Parallel Algorithms

In most design of parallel algorithms for multiprocessors, emphasis has been placed on techniques for recognizing the intrinsic parallelism of existing sequential algorithms. The aim of designing a parallel algorithm for a multiprocessor is to be able to distribute the program over 'N' computers, thereby resulting in an N-fold increase in speed, ideally. Unfortunately, this turns out to be an impossible task because of several factors. There may be problems with the hardware configuration (the interconnection of processors used may not be suitable), the algorithm design (some algorithms have more inherent parallelism than others), the cost of communicating between processors, the synchronization of processors (some processors may be idle, waiting for other processors to finish) and the amount of information shared between processors, among others.

Parallel implementation works better on computationally expensive programs. Chess program is complex and computationally very expensive and therefore is a good choice for the study. Previous works [SCHA84] [MARS85] have identified the factors responsible for the poor performance of parallel chess programs on tree machines. The degradation of performance has been attributed to three costs: search overhead (SO), communication overhead (CO) and synchronization overhead (SY). The total overhead constitutes the total loss in system performance. Search overhead is the cost attributable to the extra nodes searched in a game tree by the parallel version of the program as compared to the sequential version. Communication overhead is the additional cost incurred by the parallel programs spending time to send messages back and forth between processors. This is dependent on the software protocols and the medium of communication.

Synchronization losses are attributable to the time spent in scheduling / synchronizing processors. This arises from the idling of processors.

Schaeffer *et al.* [SCHA84] among others, have identified synchronization overhead to be the major contributor in the performance degradation of the system. Synchronization overhead causes a corresponding deterioration in performance with an increase in the number of processors. Thus, a solution to the synchronization problem has to be found to obtain appreciable gains with the addition of processors. The synchronization losses in the parallel implementation of the alpha-beta algorithm occur because of processors idling. This thesis proposes a scheme to reduces these losses.

1.4. Overview of the Thesis

This thesis explores the issues raised by the design and analysis of parallel algorithms for a multi-processor system. This thesis specifically deals with the design issues that affect a parallel implementation of the alpha-beta search algorithm, with emphasis on a tree decomposition scheme that is intended for use on ordered trees. Various types of overhead involved in parallel algorithm implementation are analyzed and a scheme is proposed to reduce the synchronization losses.

The first three chapters give the necessary background information on multiprocessor architectures, tree search algorithms and the hardware / software details of the system used to obtain results for the thesis. A virtual tree machine implementation using SUN* workstations is described in the third chapter.

Chapter Four deals with the performance of the program and identifies the

* SUN is a trademark of SUN Microsystems, California.

various kinds of overheads that degrade the performance. Experimentation results on the performance of a parallel chess program - Parabelle, using a loosely coupled network of SUN workstations, are presented. An analytical model for a multiprocessor implementation of the alpha-beta algorithm [MARS85] is briefly discussed to determine the value of various constants involved. Analysis of the results obtained for this model highlight some features of the parallel chess program.

Analysis of the synchronization overhead problem and possible solutions to reduce it are discussed in detail in Chapter Five. A scheme to reduce the synchronization overhead by deploying idle processors to help the ones that are working, is presented. Results of the experiments done to find the various costs involved in employing such a scheme are discussed.

The last chapter presents conclusions from the results obtained for this thesis and gives some suggestions for possible future work.

Chapter 2

MULTIPROCESSOR SYSTEMS AND PARALLEL TREE SEARCH ALGORITHMS

This chapter contains a brief survey of MIMD multi-processor architectures and a discussion of the parallel tree search algorithms.

2.1. Multiprocessor Architectures

Surprisingly few different multiprocessor designs have actually been built. Not much information is available on their use - a contrast to the abundance of descriptions of possible architectures. Not many multiprocessor operating systems have been implemented. In most cases the adapted uniprocessor operating systems provide acceptable performance when executed on few processors, perhaps two to four. But, because of the coarse synchronization used, most such systems would have unacceptable performance if executed on 10 or more processors. A brief description of multi-processor machines recently built will highlight the approach taken by some research teams.

2.1.1. C.mmp and Cm*

Carnegie-Mellon University has produced two MIMD machines, namely C.mmp and Cm*. C.mmp [WULF72] is composed of up to 16 processors that may be connected to a maximum of 16 shared memory modules through a 16×16 cross-point switch. An interprocessor bus connects the processors and allows any one processor to interrupt any other. Another system that is a natural extension of C.mmp is Cm*. The Cm* multiprocessor is composed of 50 computer modules, each consisting of a DEC LSI-11, a standard LSI-11 bus, memory and devices [SWAN76]. All primary memory in the system is accessible to each processor. The

main problem with these machines is the contention for the shared memory. Interprocess communication and other functions take place only through the shared memory. Therefore, the speed of accessing the shared memory may become a possible bottleneck.

2.1.2. ZMOB : A Mob of 256 Z-80 Processors

A versatile multiprocessor system with a distributed operating system is being built at the University of Maryland - ZMOB [TRIG80]. It will be a 256 Z-80 based multiprocessor system that taxonomically falls midway between MIMD and a network of processors [RIEG79]. A prototype consisting of two processors has been built and is functional, but a full system of 256 processors is yet to come. Each of its 256 processors is a completely autonomous computer with enough memory to do significantly independent high level computation asynchronously from other processors. Processors are intimately linked by a high speed interconnection system - the "conveyor belt", which is a 48 bit wide, 10 MHz synchronous shift register ring. Each processor has associated with it a 'mail stop' that allows it to send and receive data via the conveyor belt. The conveyor belt runs faster than the processors, and this enables them to communicate with each other at full speed without contention.

ZMOB will use a distributed operating system with a host controller (a VAX† 11/780). The host will handle most I/O, store data files, and load programs into the ZMOB. The operating system gives the user the image of using an ordinary UNIX‡ system but with truly parallel execution.

---

† VAX is a trademark of Digital Equipment Corporation.
‡ UNIX is a trademark of AT&T Bell Laboratories.

ZMOB differs from Cm* in three significant ways. First, it possesses a larger number of processors each roughly equivalent in power to a Cm* processor. Second, all the memory is distributed equally among the processors (there is no high speed shared memory), giving ZMOB its network flavor. However, a large shared central memory can be connected as an additional processor on the conveyor belt. Third, the interprocessor communication is handled by a non-hierarchical, contention free logic that allows all 256 processors to use the communication system simultaneously with no conflicts. ZMOB can support 128 pairwise processor conversations and broadcasts from one processor to all others or a subset of others equally well.

2.1.3. Distributed Multi-processor Systems

The term distributed computing has been used so widely over the past few years that it is difficult to decide on a generally accepted definition of the term. A reasonable definition is a collection of individual computers connected together by a local or wide area network so that by communicating among themselves as well as operating independently they can share resources and can collaborate in varying degrees for the solution of suitable problems.

Distributed multi-processor systems are therefore networks where the nodes are the processors and the links between them are any of the numerous interconnection systems (mainly local area networks). In this configuration each processor is an independent processing unit and resource sharing occurs through inter-process communication over the link. Virtual Tree Machine is based on the concept of distributed multiprocessor system and a processor tree architecture.

2.1.4. Processor Tree

A processor tree [FISH80] consists of processors (the nodes of the tree) and communication lines (the branches of the tree). The successors of a node are its slaves, while the predecessor of a node is its master. The terminal nodes are ones that have no slave, while the root node has no master. Thus, communication in this processor tree is directly with the master processor or a slave processor. The slave processors may not communicate directly among themselves. This gives a central control to the master, where important information can be collected, processed and transmitted to the respective slaves. The master processor performs the task of assignment and scheduling of work to the processors. In contrast to Cm* and C.mmp, the interconnection requirements are limited and independent of the total number of processors, and also the number of processors are extensible in a simple and regular fashion. The processor tree size can be increased to accommodate more processors by either increasing the width and/or depth of the tree. However, search strategies and processor scheduling / synchronization mechanisms used will govern the choice of proper depth / width of the tree.

2.2. Sequential Tree Search Algorithms

Game playing algorithms are programs that build and carry out searches of possible move sequences on large trees. The possible move sequences for a given position in a game can be represented by a game tree. The nodes of the tree correspond to the positions and the branches represent the moves. All nodes other than the leaf nodes of the tree are known as interior nodes. The branching factor of an interior node is equal to the number of branches leaving that node. A node is at *depth* K if it is K moves, or K *ply*, from the root. The goal of a game tree search

at a node is to determine the best score achievable from that node against an opponent who similarly chooses moves that yield his best score. An algorithm which is efficient in doing such a search is the alpha-beta algorithm [KNUT75]. It employs two bounds, alpha and beta, to control the size of the search. The algorithm evaluates the game tree at reduced cost by ignoring subtrees that cannot affect the final value of the root node. Such subtrees are said to have been *cutoff*. The interval enclosed by (alpha, beta) is known as the alpha-beta *window*. Alpha-beta algorithm is effective when the score of the root node lies within the window, and this will always be the case if the window is from $+infinity$ to $-infinity$. However, the narrower the initial window the faster the search, thus it is better to use narrow windows. In *aspiration searching*, windows are initialized to (V-e,V+e), where V is an estimate of the minimax value and e the expected error [MARS81a]. Aspiration search on position p has three possible outcomes. Either the search *fails high* (i.e. score $>$ V+e) or *fails low* (i.e. score $<$ V-e), otherwise it succeeds. If it succeeds, the true score of p is found. Search that fails must be repeated with a window that actually encloses the value for p.

A variation to the alpha-beta algorithm is the Principal Variation Search (PVS) [MARS82]. PVS relies on the fact that any knowledge about game trees can be used to pre-order the move list and hence favorably bias the shape of the game tree. The PVS algorithm gives most attention to the search of the first move, alternatives to the first move are assumed inferior until proven otherwise. The first move or the principal variation is examined with a wide window (alpha, beta). The search of the first move or the best candidate gives one the score on which the rest of the search can be based. The rest of the moves are examined with a *minimal window* of (alpha, alpha+1). Any move (subtree) searched with such a zero-width (minimal) window will necessarily fail. Most moves are easily refuted, and hence

most searches will *'fail low'*. This is so because of the preordering of moves. Whenever the search *'fails high'*, it suggests that a better move has been found. This *'better move'* should then be searched again, but now with an appropriate window. The search will give a new bound to be used for further moves. Now this move becomes the *'best candidate'*.

Certain enhancements can be used to improve the performance of the alpha-beta search. Progressive deepening [GILL78], transposition tables [MARS82], refutation tables [POPO83] and history heuristics [SCHA83] have proved useful in chess programming. The next section describes the parallel tree search algorithms.

2.3. Parallel Tree Search Algorithms

In a tree search, increasing the depth of the tree searched vastly improves the playing ability of the given program [THOM82]. Thus, it is tempting to increase the search depth to increase one's playing ability. When the depth of tree search is increased, the number of nodes to be searched increases drastically. For an optimal (minimal) alpha-beta tree search algorithm, the number of leaf nodes in a tree of depth 'd' and width 'w' is given by [SLAG69]

$$N_d = w^{\lceil \frac{d}{2} \rceil} + w^{\lfloor \frac{d}{2} \rfloor} - 1$$

where $\lceil \frac{d}{2} \rceil$ and $\lfloor \frac{d}{2} \rfloor$ represent the first integer not less than and not greater than $\frac{d}{2}$ respectively. Tree searches in chess programs for a reasonable depth search take an enormous amount of time. This search time can be reduced by multiple processors. However, the alpha-beta algorithm used for tree search is inherently sequential, and hence the application of parallelism to tree searches is non-trivial. Many radically different approaches have been proposed for employing parallelism. Among them are parallelism in primitive operations, parallel aspiration search,

mandatory work first method and tree decomposition method.

Game tree search programs widely use the basic operations of *move generation* and *terminal node evaluation*. Both these operations are potential sites for special purpose multi-processors, particularly in chess. In parallel chess move generation [CORA76], and parallel terminal node evaluation, cooperation among processors occurs at a low level and requires highly specialized interconnection mechanisms. For shallower depth searches, the move generation at the root node is substantial and thus a potential site for parallelism. It does not involve any complexity because the operation takes place at the root node and it is easy to use some specialized hardware to do the move generation in parallel.

As mentioned before, the alpha-beta algorithm performs better on a restricted window, where the window was defined to be the interval between the alpha and beta bounds. In the parallel aspiration search with N processors, N disjoint windows are searched simultaneously. If the search fails, the new window is redivided and the search is restarted with all the processors. The searches on the disjoint windows are independent, thus the searches take place concurrently without the need for a complex communication scheme. The speedup achieved by the parallel aspiration search is bounded, where speedup is defined as the ratio of the time taken by the best sequential to the parallel version of the program. Even for an optimally ordered tree search, there is a minimal tree that needs to be examined. Thus, regardless of the number of processors, one cannot achieve speedups beyond a fixed maximum. Taking processor idleness into account, it appears that this method is suitable only for the 2-3 processor case. Even with infinite processors a typical bound on the speedup is 5-6 [BAUD78].

Another approach to parallelization is the **mandatory work first method** [AKL82], which decomposes the search tree and assigns subtree searches

dynamically, i.e., based on the current status of the overall tree search. Here the processes (nodes to be searched) are kept in a priority ordered set. The highest priority process gets the next available processor. The priority ordered set is reordered at each step. The priorities can be assigned to take care of processor idleness also. However, the implementation is complex and poses several problems. In particular, a means of selecting and suspending processes must be found, one that does not involve much communication and storage overheads. In contrast to this method, the tree decomposition scheme generates and assigns subtree searches in a fixed, pre-determined way.

The tree decomposition scheme [FISH81] involves assigning separate subtrees of a game tree to individual processors. A master-slave configuration is best suited to this scheme. The master process will generate all legal moves at the root node and will assign moves to available processors. The master processor passes the search depth and alpha-beta window for the search along with the move. Baudet [BAUD78] concluded that a large performance improvement can be obtained by combining parallel aspiration search with tree decomposition. This approach suggested by Baudet is the one that forms the design framework of the algorithm used for this thesis. This approach however is not without its disadvantages, which are discussed in detail in the following chapter.

2.4. The Parallel Minimal Window Search

The Parallel Minimal Window Search (PMWS) [MARS85] is a tree decomposition scheme that is done only after the first path along the Principal Variation has been searched. A processor tree is used for the implementation of this algorithm.

Here, the principal variation search gives the score on which the rest of the subtrees search is based. Each subtree is assigned to a separate processor, which searches this with a minimal window. When a processor finishes a search, it gets the next move from the movelist. When no moves remain to be assigned, the slave processor will become idle. This idle period contributes to the overhead in a parallel implementation.

The following observations can be made from a closer look at the PMWS algorithm:

a) For a processor tree of length one (i.e. the slave processors do not have any slaves to themselves), the degree of concurrency is limited to the branching factor of the search tree.

b) During the minimal window search of subtrees at the root node, some moves may *'fail high'*. This represents a new principal variation or candidate. This *'new candidate'* is searched by a single processor as opposed to the *'first candidate'* which is searched by all processors.

c) In a processor tree of length one, the slaves get assigned moves from the movelist at the root node. When the movelist becomes empty, the slaves on completion of their search start idling - waiting for all other processors to finish their search. This is a major disadvantage with a processor tree of length one.

With some complexity involved, one can overcome these drawbacks *b* and *c*. Drawback *a* is not yet a practical drawback, because in chess game trees the average branching factor is about 35, that means that one is limited by a processor tree fanout of 35 (i.e. the master has at most 35 slave processors). Now researchers are striving to find efficient solution to parallel tree searches for processor tree fanout of 2-6 processors.

Drawback *b* can be overcome by interrupting all the processors and abandoning all current tree searches, and then employing all the processors to the search of the *'new candidate'*. This is then exactly like the search of the *'first candidate'*. The last drawback mentioned, that of idleness of processors, causes significant losses and is the major concern in this thesis. A detailed analysis and solution to this problem is presented in chapter 5.

Chapter 3

SOFTWARE AND HARDWARE SYSTEM CONFIGURATION

This chapter discusses the implementation of a parallel chess program using the PMWS algorithm for tree search. To identify various issues involved with a multiprocessor system and parallel tree searching algorithms, a version of a parallel chess program - Parabelle [POPO83] has been revised on a loosely coupled network of SUN workstations and VAX/UNIX processors.

A version of Parabelle has already been implemented on a multiprocessor system consisting of four SUN (MC 68000 based) processor boards, with software support provided by a VAX/UNIX system [POPO83]. However, the results for this thesis have been obtained by implementing the parallel chess program (Parabelle) on a more powerful and flexible system. This multi-processor system is based on Ethernet* - a bus oriented local area network (LAN), that couples the various SUN workstations and VAX/UNIX processors. This system provides for a much simpler inter-process communication, and ease of set-up and reconfiguration of the processors.

The program implemented for this thesis study runs on a virtual execution environment. This environment is called a "Virtual Tree Machine" (VTM). A virtual tree machine consists of a fixed finite network of processors together with a suitable algorithm for deciding when and where to solve each subproblem. Simulation results [BURT84] suggest that a VTM can execute large programs with a high processor utilization and modest memory requirements.

Parabelle analyzes chess positions using a processor tree network. In a

* Ethernet is a trademark of Xerox.

processor tree, physical processors are connected in a tree structured fashion such that each node (processor) can only communicate directly with its parents and children nodes. In a VTM, physical processors are replaced by processes and hard-wired interconnections are replaced by virtual communication paths. The VTM* configured in the Department of Computing Science at the University of Alberta consists of SUN workstations and VAX/UNIX machines. The configuration provides the full capabilities of the UNIX operating system (4.2bsd version) to each node in the "processor tree". The system facilities provided include file handling, debugging and performance monitoring.

A "tree machine" is created from the topological map of the desired system before actually giving control to Parabelle. It is possible to make more than one node reside on a physical machine. All processors are linked to Ethernet that imposes no preferred configuration, thereby allowing one to create a processor tree of any desired depth or fanout.

3.1. Virtual Tree Machine Implementation

To understand the concept of a Virtual Tree Machine and its implementation, let us consider an example. Suppose we need to create a processor-tree machine as specified in Figure 3.1. We will use four workstations to implement this processor tree. First, we need to associate nodes N1, N2, and N3 of the processor tree to physical processors (SUNs). Assuming the names of the four SUN processors to be ***sunshine,***

* The design and implementation of the VTM system has been done by Marius Olafsson [OLAF85]. The inter-process communication routines for the VTM have been built from the communication primitives provided by 4.2bsd version of UNIX.

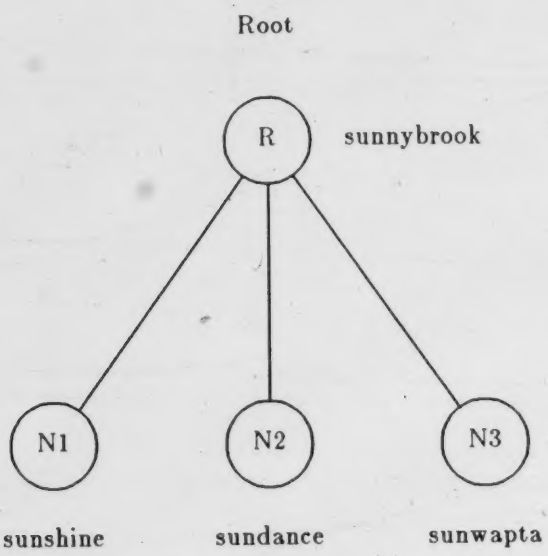


Figure 3.1 A Sample Processor Tree with SUN Processors

sunnybrook, *sunwapta* and *sundance*, we can make the assignment as follows:

The Root (R) on Sunnybrook
Node N1 on Sunshine
Node N2 on Sundance
Node N3 on Sunwapta.

Once an assignment is done, we can create a "**configuration file**"** The configuration file will contain the names of the machine involved, the number of slaves each machine has and the location of the object file. It also has parameters for specifying the name of the input file, the error output file and an option to produce diagnostics. The above assignment will produce a configuration file containing the following information:

```
sunnybrook;3;0
        sunshine;0;0;slave;;out1;err1;
        sundance;0;0;slave;;out2;err2;
        sunwapta;0;0;slave;;out3;err3;
```

Each entry in the configuration file corresponds to a node of the processor tree. In the N1 node entry, the name of the machine is 'sunshine', the parameters "0;0" show that node N1 has no slaves and that diagnostics are disabled. The parameter 'slave' is the name of the object file, 'out1' and 'err1' are the names of the output file and the error output file, respectively. The capability of specifying the name of the object file to be executed on slaves eliminates the need to start the slave processes on the slave processors individually. More details on the configuration file are laid out in the technical report on VTM by M. Olafsson [OLAF85]. The above configuration file results in the hardware configuration of Figure 3.2.

A VTM node-server running on each processor involved in this processor tree controls the configuration of the system. When the master processor is invoked on sunnybrook, the configuration file residing on sunnybrook is read and an internal

** Syntax designed by Marius Olafsson [OLAF85].

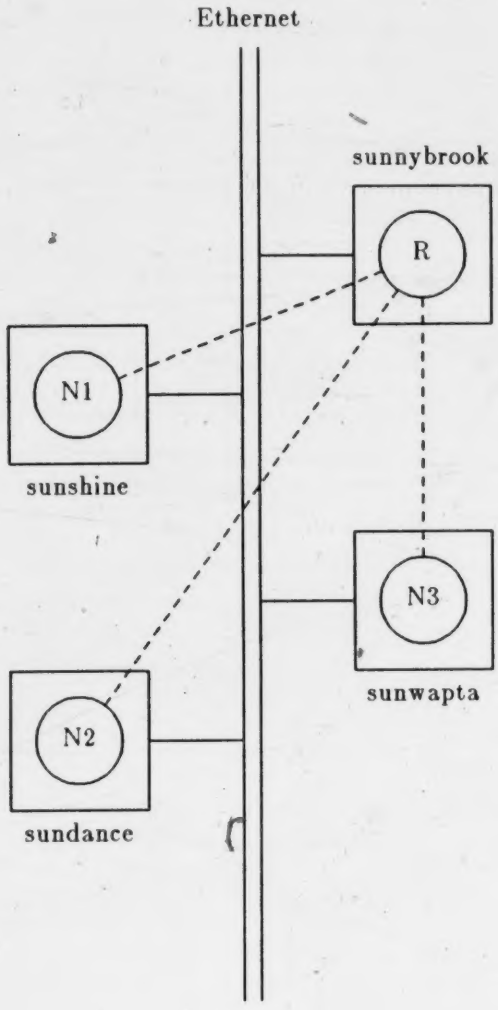


Figure 3.2 Processor Tree Hardware Configuration

representation of the processor tree is created. The master process sees that it has 3 slaves, and passes the representation to the three node servers on sunshine, sundance and sunwapta. When the slave processors see that they have no slaves themselves, they return a success message to the master process. The control is then passed to Parabelle.

This virtual tree machine has the advantage of cost saving, ease of debugging and performance monitoring. The system provides for easy reconfiguration and enables one to try different processor tree depths and fanout without the need of any hardwiring and physical re-orientation of the system. A disadvantage of this may be the sharing of the resources with other users on the system, thereby preventing accurate timings. To avoid causing inconvenience to other users during the period of experiment, the tests for this thesis were run at night time.

3.2. Inter-unit Communication

In the parallel implementation of the PMWS algorithm for the chess program, the communication between processors occurs in various places. The parallel implementation of the chess program used for this thesis has the following major inter-unit communications [MARS85] :

a) inter-node communication
b) inter-level communication
c) inter-iteration communication
d) interactive message communication.

Inter-node and inter-level communications are associated with PV node searches. In the PMWS algorithm, the master assigns a move to a slave and the slave in turn returns a value after searching the tree. This communication between the master and slave is referred to as inter-node communication. Inter-level communication is required to send message to the master after completion of a PV node search.

Inter-iteration communication occurs when progressive deepening is employed. These communications account for the part of the overhead in a parallel implementation.

3.3. Inter-process Communication

The Virtual Tree Machine implementation and Inter-unit communication described above are based on the Inter-process communication primitives provided by the 4.2bsd version of UNIX [LEFF83]. These communication primitives provide for a reliable two-way communication path. The networking protocols used by the communicating processes are transparent to the user.

The interprocess or VTM internode communication is facilitated by high level calls of s_send (send to slave/descendant), m_send (send to master/parent), s_receive (receive from slave/descendant) and m_receive (receive from master/parent). These calls are used by each node in the VTM to communicate with its descendant (slaves) and its parent (master). The slave nodes are numbered 1 to m (where m is the number of descendants of the parent node). The parent node may be considered as descendant number 0. These calls enable a process to transfer any specified number of bytes to a parent or slave process, with the option of interrupting the destination process on arrival of the message.

A process may have a critical section, where interrupts from all other processes may have to be disabled. Such critical sections are handled by the VTM interrupt routines, that enable the process to manage interprocess signals. Each VTM node is capable of interrupting its descendant and its parent. Receiving nodes are also capable of disabling (ignoring) and enabling interrupts caused by incoming messages and can hold up to one interrupt pending. These interrupt routines are used to manage critical sections of code in the Parallel Minimal Window Search

algorithm. Other useful VTM communication routines include routines to check the status of VTM communication channels, routines to get information about the processor tree and search primitives and routines to dynamically reconfigure the Virtual Tree Machine.

Before the introduction of these communication primitives in the 4.2bsd version of UNIX, the communication between processors was slow and cumbersome. These communication primitives have provided an elegant abstraction from the raw networking protocols. The results presented in the next two chapters have been obtained from the implementation of PMWS algorithm on such a system.

Chapter 4

PERFORMANCE ISSUES

The design of multi-processor systems and parallel algorithms is directed towards a common goal of achieving fast and efficient processing. One success/failure measure for a parallel algorithm is the "speedup" achieved. The objective behind the design of parallel algorithms is to be able to achieve speedups that increase linearly with the addition of processors. To put it in simple terms, ideally one would like to get a speedup of 'N' by using 'N' processors in the system. Unfortunately, this becomes an impossible goal to attain [POPO83] [SCHA84]. It is difficult to achieve even something close to it. There are some inherent problems with parallelism and the alpha-beta algorithm in particular. It is the aim of this chapter to go into the issues involved in the design of parallel algorithms and identify factors that are responsible for the degradation in the performance of parallel algorithms. Also, experimental results for an analytical model developed by Marsland are discussed to highlight some features of the PVS algorithm. The next section presents a model that will be used to define the various overheads.

4.1. Overheads

Consider the case of a N processor system. The master process maintains a list of valid moves, it assigns a move to each slave processor. When a slave finishes its assigned move, it gets another move assigned from the move list. Finally the move list will become empty and the slaves cannot be assigned any more moves. Now, when a processor finishes its search, it has no assignment and thus becomes idle. For a processor tree of length one, up to $(N-1)$ processors will become idle,

waiting for the N^{th} one to finish. To build a simple model, consider a four processor system. Figure 4.1 shows the working time and the idle time for each of the processors. The solid line depicts the working time and the dashed line shows the idle time of a processor. One can define the following variables :

t_{seq} = time taken by one processor to search the tree
t_1 = working time for the 1^{st} processor
t_2 = working time for the 2^{nd} processor
t_3 = working time for the 3^{rd} processor
t_4 = working time for the 4^{th} processor
I_1 = idle time for the 1^{st} processor
I_2 = idle time for the 2^{nd} processor
I_3 = idle time for the 3^{rd} processor
t_{idle} = total idle time of the system

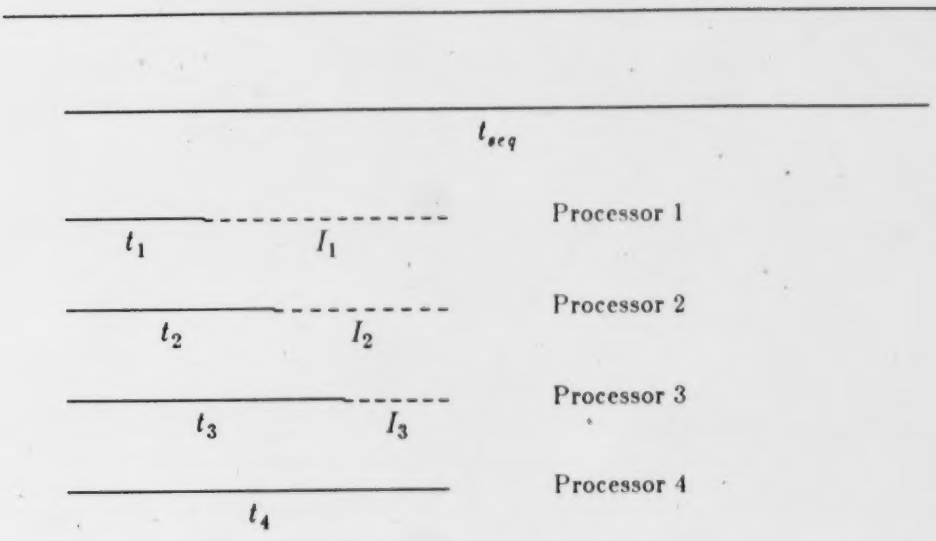
These notations will be used throughout the rest of the thesis. Resources of all the processors are tied up until the last one finishes. The real time and resource time mean the same thing here.

A brief introduction to overhead was presented in the first chapter. The three types of overhead namely search overhead, communication overhead, and synchronization overhead constitute the total loss in system performance. This overall loss in time is termed time overhead (TO), which can be defined as the percentage loss in speedup when using N processors compared to the ideal speedup, and this can be expressed as,

$$TO = \frac{(Time\ using\ N\ Processors) * N}{Time\ using\ 1\ Processor} - 1 \qquad (1)$$

Considering a four processor system and using the notations described in Figure 4.1, one can get :

Time using four processors = $4t_4$
Time using one processor = t_{seq}



$$t_{idle} = I_1 + I_2 + I_3$$

Figure 4.1 Working and Idle Time of Processors

Therefore, TO expressed as a percentage of t_{seq} is given by

$$TO = \frac{(4t_4 - t_{seq})}{t_{seq}} \times 100 \quad (2)$$

4.1.1. Communication Overhead

Communication overhead is the time lost by parallel programs in sending messages back and forth between processors. It is a function of the hardware and software protocols used to connect the computers and the amount of communication involved. In chapter 3, there was a mention of the various types of inter-unit communications that take place between processors. The total of such communications constitute the overall communication overhead and can be estimated by counting the number of messages send back and forth between processors. In the VTM implementation using the Ethernet connections, the communication time is mainly the setup time for constructing the message and setting up the processors to send and receive the messages. The actual byte transfers are fast and take negligible time in comparison to the message setup time. Experiments on the VTM show that the message setup time is about 9.2 msec and 4 bytes of message transfer between processors takes an additional 0.1 msec only. Even 32 bytes of message transfer takes just 0.5 msec. This suggests that one can group message transfers to send as many bytes of message at a time as possible to minimize message setup time and hence reduce communication overhead. Communication Overhead can be expressed as :

$$CO = \frac{No.\ of\ messages \times time\ for\ one\ message}{t_{seq}} \quad (3)$$

4.1.2. Search Overhead

Search overhead is defined to be the cost attributable to the extra nodes searched in the parallel version of the program as compared to the sequential version. In a sequential alpha-beta algorithm, the "complete" information is managed by a single process, thus at any point in the search the program has access to all the previously computed results. It means that it has the best alpha bound and all the transposition table entries from previous iterations. On the other hand, in a parallel environment, subtrees are searched on different processors and the total sharing of information is not practical. If one has to provide each processor with the complete information, as that available to processors in the sequential case, then all the processors have to continually share their knowledge with each other. Every time a slave processor finds a better score, it has to communicate to the master and the master in turn has to send this value to all the other slaves. Obviously, this increases the amount of communication required between the processors and hence the communication overhead. On the other hand, if each processor takes little or no advantage of results from other processors, it is inevitable that extra nodes will be searched because of the possible lack of the best alpha bound and the transposition table entries from previous iterations. Here one encounters a trade-off between increased communication costs and extra nodes searched. The search overhead can be approximated by knowing the extra number of leaf (terminal) nodes searched by the parallel algorithm. The time spent in searching is proportional to the number of leaf nodes searched. The search overhead can be estimated by:

$$SO = \frac{[leaf(n) - leaf(1)] * av.\ time\ to\ evaluate\ a\ leaf\ node}{t_{seq}} \qquad (4)$$

Since, time to search is proportional to the number of leaf nodes searched, one can get

$$SO = \frac{[leaf(n) - leaf(1)] * av.\ time\ to\ evaluate\ a\ leaf\ node}{leaf(1) * av.\ time\ to\ evaluate\ a\ leaf\ node}$$

$$= \frac{[leaf(n) - leaf(1)]}{leaf(1)} \quad (5)$$

where leaf(n) is the number of leaf nodes searched when using n processors.

4.1.3. Synchronization Overhead

The synchronization loss can also be termed idling loss. Synchronization overhead is caused whenever a processor becomes idle, and waits for the other slave processors to complete their search. In the parallel implementation of the PMWS algorithm, the loss occurs at each PV node along the path of the first candidate (principal variation). As explained earlier, processors become idle when no moves remain to be searched. In the worst case, in an N processor system, N-1 processors might be waiting just for the Nth (the last processor) to finish its work (move). Synchronization overhead can be calculated by explicitly measuring the idle times of processors. From Figure 4.1, one can estimate the idle losses,

$$Total\ Idle\ Time\ (t_{idle}) = I_1 + I_2 + I_3 \quad (6)$$

$$SY = \frac{I_1 + I_2 + I_3}{t_{seq}} \quad (7)$$

Synchronization overhead can be approximated by using the known values of time overhead (TO), communication overhead (CO) and search overhead (SO). Synchronization losses account for the remaining part of the total losses.

The goal behind the design of parallel algorithm is to maximize program efficiency, in this case the speedup. This means that one needs to minimize the overhead. Therefore, it is important to analyze the results of the parallel implementation and discover means of reducing any significant losses. As evident from the discussion of search overhead, communication overhead is inversely proportional to the search overhead. A decrease in search overhead is at the expense of an increase in communication, and hence a balance has to be found between the various losses.

Before the results for the performance of the PMWS algorithm and the various types of overhead are presented, it is worthwhile to look at the analytical model proposed by Marsland [MARS85] and discuss some experimentation results for the model. The results highlight some features of the PMWS algorithm.

4.2. Analytical Model

The analytical model is useful in modeling an alpha-beta search using the PMWS algorithm. For analysis, the search tree is assumed to be a uniform game tree of width $'w'$ and depth $'d'$ in which $(1+g)$ branches are searched at cut-off nodes. A processor tree of length one and fanout $'f'$ is employed to search the trees. An estimate of the search time for such a tree provides a lower bound on the alpha-beta search tree. In the analysis, Marsland arrives at the following equations for the number of leaf nodes in a minimal alpha-beta tree :

$$T_{d,f} = k + T_{d-1,f} + \left\lceil \frac{(w-1)}{f} \right\rceil B_{d-1} + (w-1)\frac{(f-1)}{f}(s+t_1) + (f-1)(s+t_2) \qquad (8)$$

with

B_d = time to do a zero window search to depth d.
$T_{d,f}$ = time to search a minimal tree to depth d.
d = depth of search.
e = time to evaluate a single move at a leaf node.
k = time to create node and generate move list.
s = setup time in handling a message.
t_1 = time to transmit an inter-node message of length b_1 bytes.
t_2 = time to transmit an inter-level message of length b_2 bytes.

where

$$T_{0,f} = e$$
$$B_d = 2k + w\,B_{d-2}, \quad \text{for } d > 1,$$
$$B_0 = e$$
$$B_1 = k + e$$

These equations were derived for a minimal alpha-beta tree. Marsland assumes that a typical alpha-beta tree is approximated by a uniform alpha-beta tree with (1+g) branches expanded at each cut off node, and derives the following equation for the time taken to evaluate the leaf nodes of a uniform tree (for d=2m+1),

$$E_{2m+1,g} = (w-1)e\,[\sum_{i=0}^{m}(1+g)^i\,w^i + \sum_{i=0}^{m-1}(1+g)^{i+1}\,w^i - 1] + we \qquad (9)$$

These equations form the basis for the result and deductions that follow in the next section.

4.3. Analysis of Experimental Results for the Analytical Model

The equations derived for the analytical model can be used for the estimation of the number of nodes searched in a game tree and the search time for a designated number of processors and the depth of search. Experiments were conducted on the chess-board positions from the Bratko-Kopec Test [BRAT82], working with a 5 ply tree search. The experimental results are presented in Table 4.1. The results obtained from the experiment can be used to estimate the values of the unknown

Node Counts for 5-ply Search						
Board	enodes	lnodes	qnodes	cutoffs	cutnodes	fullnode
B	17196	9468	25485	6673	4206	1003
C	17413	7000	23910	6082	4093	866
D	3887	2317	36426	6653	3692	1179
E	33407	18092	80264	10909	6660	2415
F	942	593	9970	1686	1412	561
G	32900	15980	39166	7178	4408	1202
H	179	174	3753	2001	731	217
I	39600	20442	54930	8416	5596	1639
J	53928	23714	59883	9399	5908	1552
K	31153	12832	42907	6639	4124	1256
L	19353	9704	28556	5784	3661	887
M	16791	9048	59830	7009	4677	1688
N	2736	1475	23039	2887	2718	909
O	12408	4495	20123	2062	1915	953
P	19808	12511	43991	11759	4122	1246
Q	31437	14663	40216	13787	7941	1288
R	29920	16902	66992	9221	5523	1713
S	24866	11667	33183	7355	4670	1114
T	74198	43069	108299	23747	12651	2966
U	78492	30156	75699	13054	7722	2002
V	9387	4870	30013	4187	2984	1038
W	89944	43213	113924	60266	13516	2128
X	6246	3691	58595	5091	4723	1542

where,

qnodes: terminal node count for a quiescent search.

enodes: total count of moves that are evaluated at qnodes.

lnodes: no. of qnodes at which at least one move is evaluated.

cutoffs: no. of moves where a cutoff occurs.

cutnodes: no. of nodes at which the moves had a cutoff.

fullnode: no. of nodes that are fully evaluated (no cutoffs).

Table 4.1 Node counts for a 5-ply Search / Single processor

constants for the equations (8) and (9) presented in the previous section.

To summarize the results from Table 4.1, we have the following average values :

- average number of terminal nodes evaluated (Qnodes) = 46919
- average number of capture moves at a leaf (terminal) node = 2
- average number of moves examined at a node = 1.9
  thus, branching factor = 1.9, and g = 0.9
- average width of the tree = 35
- the percentage of nodes where a cutoff occurs to the nodes that are fully evaluated = 80%

The timing for these experiments was done by the system timer that has a granularity of $\frac{1}{60}$ sec. From the above average values and timing on these, we have the following values for the constants 'k' and 'e' in equation (5) and (6),

e = average time to evaluate a single move at a leaf node
= 0.032 secs.
k = average time to create node and generate movelist
= 0.02 secs.
s = average setup time in handling a message
= 9.2 msecs.

Some observations can be made from these results. First, the average value of branching factor is 1.9, which means that on an average only 0.9 more moves at each node are evaluated as compared to the minimal alpha-beta tree. This factor of 1.9 shows that the ordering of moves and subsequent PV search are extremely effective in causing an early cutoff. It also means that the moves in the local transposition tables are stored efficiently. If it were not for a strong ordering of moves and an efficient PV search, the program would have searched more moves before cutoffs and hence a higher branching factor. This search examines on an average 46,919 leaf nodes as compared to an expected minimal case of 44,000 for a 5 ply tree search, this also indicates that cutoffs occur early when using the PMWS search strategy on a strongly ordered game tree.

Clearly the various enhancements incorporated in the chess program are extremely effective in causing early cutoffs and hence to reduce the search times drastically. However, it does not necessarily mean that these enhancements will be equally effective in a N processor case, where $N > 1$. There are other problems that degrade the speedup in a multi-processor case as discussed before. Let us analyze the various overheads in the parallel implementation of the program on a network of SUN processors.

4.4. Results for a 5-ply Search

Results from Parabelle [POPO83] and Para-Phoenix [SCHA84] show that as the number of processors are increased, there is a degradation in the speedup that finally reaches a point where the addition of an extra processor only yields a marginal speedup. The results for experiments were obtained for a 5-ply search, while working on the chess-board positions from Bratko-Kopec test [BRAT82]. The details of these results are in Table 4.2. Figure 4.2 shows the speedup achieved by using various number of processors (1-4). Figure 4.3 shows the various overheads as a function of the number of processors.

In a multi-processor case, since the score is local to a processor, the positions seen by other processors will not be able to produce a cutoff for the processor in question. Different cutoffs will then result depending on which processor searches which subtrees. This may also result in a decrease in the number of terminal node count as the number of processors increase (as with board positions E and W), although this is not generally true. Some positions like D, F, G, H and V show bad speedups, this may be because of the larger subtree searches occuring later in the tree search.

Parabelle2: Results from a 5-ply Search									
Board	Terminal Node Count				Search Time (sec.)				Speedup
Pos.	1Pr.	2Pr.	3Pr.	4Pr.	1Pr.	2Pr.	3Pr.	4Pr.	4Pr.
B	25485	25619	25716	25796	595	312	214	172	3.5
C	23910	24045	24335	24506	1057	578	387	330	3.2
D	36426	36681	36717	40438	665	382	237	308	2.2
E	80264	81200	81202	81483	2355	1220	807	634	3.7
F	9970	10162	11009	11042	85	48	40	37	2.3
G	39166	39254	48461	49600	1155	609	612	480	2.4
H	3753	3943	4130	4042	31	19	15	15	2.1
I	54930	55892	55931	55991	1599	884	604	489	3.3
J	59883	63967	64807	70636	2243	1256	864	776	2.9
K	42907	43386	43581	43523	1182	623	439	334	3.5
L	28556	28569	28609	28641	749	376	263	209	3.6
M	59830	62088	62132	62421	974	477	324	260	3.8
N	23039	24977	25874	26406	518	291	209	171	3.0
O	20123	20386	20636	20646	460	256	180	153	3.0
P	43991	43866	44069	44283	986	542	400	346	2.9
Q	40216	41018	41188	41744	1245	632	450	379	3.3
R	66992	69206	70058	70338	1765	876	631	478	3.7
S	33183	33238	33339	33527	759	400	272	216	3.5
T	108299	108886	111112	112242	2299	1212	867	710	3.2
U	75699	77835	78142	78215	1719	869	611	472	3.6
V	30013	30213	30331	30401	1066	569	473	478	2.2
W	113924	114375	112975	113154	1570	805	631	470	3.3
X	58595	58817	58830	58927	1341	671	490	372	3.6
Aver.	46919	47722	48399	49043	1148	606	435	360	3.2

Table 4.2 Multiprocessor Results from a 5-ply Search

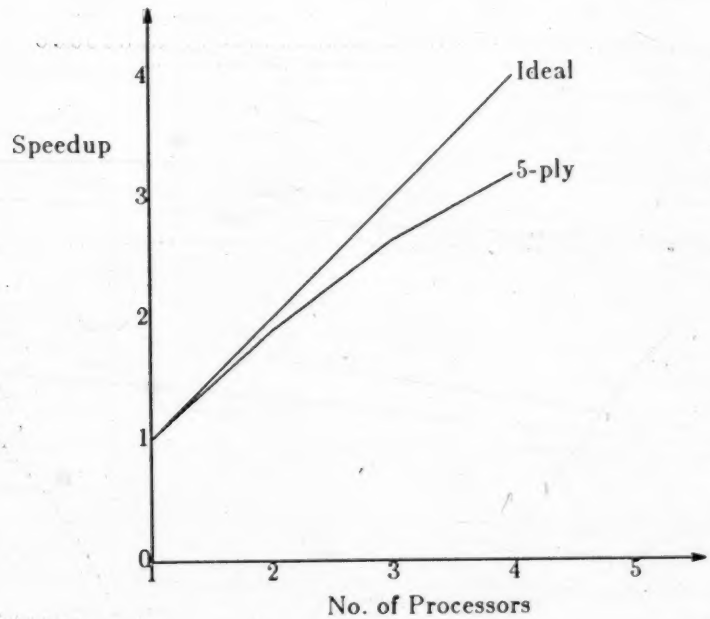


Figure 4.2 Speedup vs No. of Processors

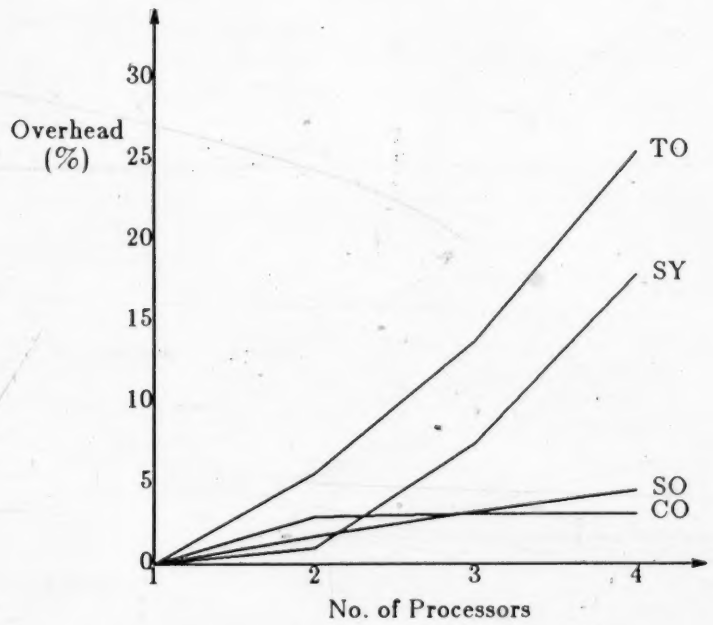


Figure 4.3 Average Overheads for a 5-Ply Search

Table 4.2 and Figure 4.2 show that the speedup achieved decreases and the overhead increases with the increase in number of processors employed. To understand the limitations of the parallel algorithm and to find ways to improve performance, it is important to analyze the reasons for this degradation in performance. To understand this and come up with some solution, one needs to look at the three major types of overhead more closely.

Using equation (2) and the average values from table 4.2 one can see that the time overhead (TO) increases from a mere 5.6% in a 2 processor case to 13.7% in a 3 processor case and increases to 25.4% in a 4 processor case. This increase in overhead is expected to be even steeper with more processors added [SCHA84].

Communication overhead, as seen from Figure 4.3, is small and is a constant percentage of the total loss. It was calculated by measuring the number of messages sent between processors and multiplying this total number of messages by the time to send one message (9.3 msec). In this experiment it accounts for about 3% loss in overhead. This clearly becomes negligible as the number of processors increase or the search depth is increased [SCHA84]. Communication overhead is low because an Ethernet (10Mbit/sec.) link is used for communication between processors. Also, the PMWS algorithm by design keeps the inter-process communication as low as possible.

It is also interesting to note that the search overhead (SO) does not increase significantly with an increase in the number of processors, this is because of the efficient search strategy (PVS and other enhancements) used. The search overhead seems to be comparable in a 2 processor case as compared to synchronization losses. This is because in a two processor case only one processor may be idle, not much time is lost in idling, and hence synchronization losses are low. But as one goes to the 3 or 4 processor case, SO levels off but the synchronization overhead increases

drastically and clearly evolves as the major overhead for more than 2 processor implementation. This is easily explained, since the greater the number of processors, the higher the likelihood that some will be idle, waiting for others to complete their search.

It can be easily visualized from Figure 4.3 that CO and SO seem to flatten out and become minor losses for the case of more than three processors. The major overhead that is responsible for this notorious behavior of reduction in speedup is clearly synchronization overhead. As evident from the graph in Figure 4.3, it prevents one from getting reasonable gains with the addition of processors. Thus, it is extremely important to look at methods to reduce synchronization losses. The next chapter analyzes a solution to this problem.

Chapter 5

SYNCHRONIZATION OVERHEAD

This chapter contains an analysis of the synchronization overhead problem and a discussion of a scheme to reduce it. Experimental results presented in the previous chapter clearly show the behavior of the various types of overhead in a parallel chess program. The three overheads - synchronization, search and communication, account for the degradation in performance of a parallel game tree search algorithm. The results show that degradation in processor utilization increases as the processor count increases, although communication and search overhead do not grow fast and seem to stabilize rapidly. When a system consisting of four processors is used, its performance is degraded. From an examination of the behavior of each overhead, it is clear that a limiting plateau will soon be reached after which the addition of a processor will yield no appreciable gain. Schaeffer predicts from his results that with 8 processors Para-Phoenix [SCHA84] will achieve a maximum speedup of only 4.4. From the results presented in the previous chapter and assuming that processor utilization drops at a uniform rate, Parabelle is expected to achieve a maximum speedup of only 4.7 with 8 processors.

A large amount of research by various research teams has gone into achieving asymptotic behavior but without any success. However, this research done by various people has clearly identified that a solution to the synchronization problem has to be found to get any appreciable gains from the addition of processors. The next section analyzes the synchronization overhead problem in more detail.

5.1. Solutions to the Synchronization Problem

Under the PMWS algorithm implemented on a multi-processor system, synchronization losses occur only at PV nodes. The root node and all other nodes in the first path of a principal variation search are PV nodes. In a system consisting of N processors and node width 'w', all N processors traverse the first path at a PV node and then split the remaining $(w-1)$ subtree searches. As the search terminates and there are no more moves to be assigned from the now empty movelist, from 1 to (N-1) processors become idle. The number of idle processors increases with the total number of processors, resulting in a higher synchronization loss. The processors become idle because of the highly variable search times for the subtree searches and also because of the imbalance between node width 'w' and the number of processors.

For example, if 'w' and N are equal and the search time does not vary, there would be no synchronization loss. But this is never the case. Because of the variable search times and/or imbalance between node width and the number of processors, there is always a synchronization loss. If one could find out the behavior of the moves in a movelist, one could schedule the moves in a way to reduce the processor idle times. This would imply extensive study of the moves and their characteristic behavior.

One can refer to Figure 4.1 to understand the problem. The processors in a four processor system are idle for time I_1, I_2, and I_3 respectively and the total idle time is given by t_{idle}. The solution to the synchronization (idling) overhead problem will consist of reducing the idle time, t_{idle}. The observation that the idling of processor occurs because a processor has to wait for all the others to finish, leads to a possible solution to this problem. The solution is to develop ways of deploying

the idle processors to assist the others that are still working.

The solution seems to be a good way of handling the scheduling of processors - the idle processors are deployed to help the busy ones. Whenever a slave processor becomes idle because it has no moves to make, it sends a message to the master processor informing it of its availability. The master processor receives the message from the slave and assigns it to assist one busy processor. The implementation of this scheme will cause an increase in communication overhead because of the extra communication involved between master and slaves. Communication is also required between the master and the busy slave to pass control of the idle slave. However, the communication over Ethernet using inter-process communication primitives provided by 4.2bsd UNIX is efficient. In chapter Four it was shown that the communication overhead is a mere 3% of the total search time, the increase in communication overhead for implementing this scheme is expected to be of this order. Thus the gains obtained by deploying idle processors should be more than the loss in increased communication. This approach forms the basis for the scheme proposed in the next section.

5.2. Scheme to Reduce Synchronization Overhead

There was a discussion on some possible solutions to the synchronization problem in the previous section. This section covers the scheme of deploying idle processors to help the ones that are still working. In a processor tree of length one, the master processor assigns moves to slave processors from its movelist. When a slave processor completes the search of the move assigned to it, the processor sends a message to the master and updates the score maintained by the master. The master assigns the next move from its movelist and passes to the slave the best score obtained thus far. This process continues for as long there is a move in the

movelist. When the movelist becomes empty and the slaves terminate their search, processors will become idle. But instead of letting them sit idle, they can be deployed to help the busy processor. Before one proceeds into detailing the scheme, a definition of the processor tree for this implementation is in order.

The VTM implementation of a processor tree requires that before the control is passed to the program, the nodes and the communication paths for the processor tree be defined explicitly. If one wants an idle processor to assist any busy processor, one would have to explicitly define communication paths between all the processors, thus resulting in a fully connected network. This will not be bad for a 4 processor case, but for 10 processors or more it will become a complex network. This can be clearly understood by trying to see what the "configuration file" would look like if one lets this happen. However, to maintain a simple processor tree structure, one can define a path to only one other processor from any slave processor. This implies that a busy processor can be helped by only one other pre-designated processor. The busy slave processor will have capabilities similar to those of the root processor. This means that it can act as a master to the newly assigned processor or function as a slave to itself. Thus, this can be modeled as a two processor case for each subtree search. Taking the case of a four processor system, Figures 5.1 and 5.2 show the processor tree and configuration file required of the current implementation of Parabelle. The four processors used in the implementation are :

sunnybrook (SB) or (R)

sunshine (SS)

sundance (SD)

sunwapta (SW)

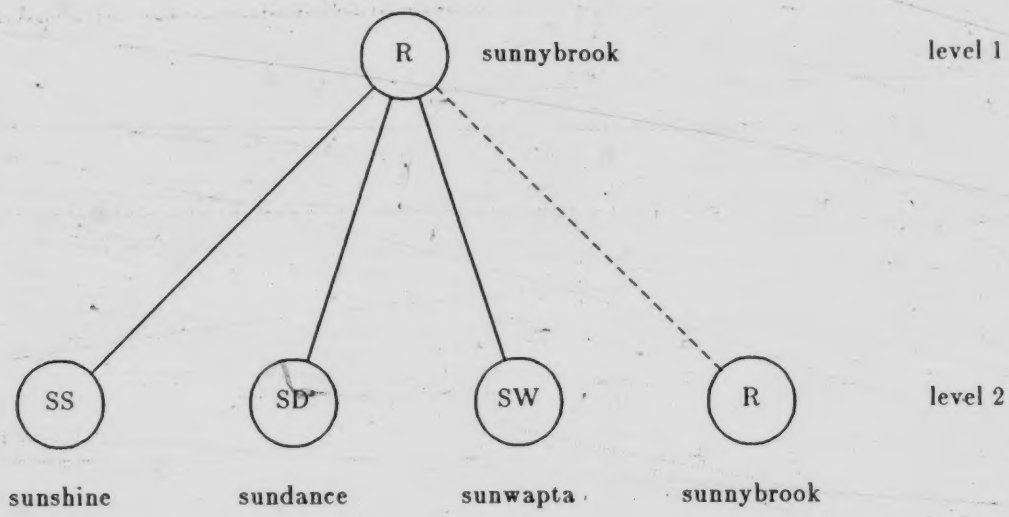


Figure 5.1 Processor Tree for Current Implementation

```
sunnybrook;3;0
     sunshine;0;0;slave;;
     sundance;0;0;slave;;
     sunwapta;0;0;slave;;
```

Figure 5.2 Configuration File for Current Implementation

The node R is also shown as a slave of the root process because the master processor software has the capability of being a master and slave at the same time. These types of connections where the node has the capability of both the master and the slave are shown by dashed lines. These connections need not be defined in the configuration file because they are embedded in the software itself. Thus, effectively there are four processors helping in the search of moves at the root node. in the search of moves at the root node.

In the current implementation (Figure 5.1), it can be seen that slave processes SW, SD and SS do not have any slaves themselves. But the scheme - to deploy idle processors to help the ones that are busy, requires that level 2 slaves themselves have slave processes. For the implementation of such a scheme, assume the following relationship for this discussion :

Processor	Can be helped by
SS	SW
SD	SB
SW	SD
SB	SS

The fully expanded processor tree and configuration file is shown in Figures 5.3 and 5.4. Here also, the nodes SS, SD, SW and SB(R) act as slave and master at the same time. The search of moves at the root node is similar to that of the current implementation of Parabelle. Algorithms suitable for doing the search at root node are given in Figures 5.5, 5.6 and 5.7. The routine *m_process* in Figure 5.7 handles interrupts from the slave processors, thus m_send and m_receive function calls in *s_search* will have corresponding s_receive and s_send calls both in *m_search* and *m_process*.

The master process maintains a structure that holds the data defined in the configuration file. This enables it to know the processors that can help each other.

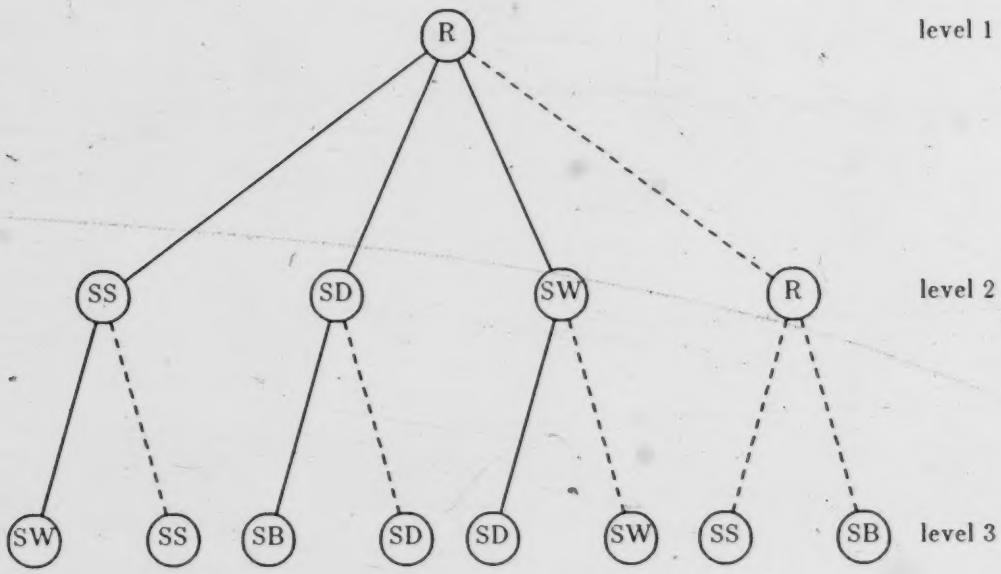


Figure 5.3 Processor Tree for Proposed Implementation

```
sunnybrook;3;0
        sunshine;1;0;slaveR;;
              sunwapta;0;0;slaveS;;
        sundance;1;0;slaveR;;
              sunnybrook;0;0;slaveS;
        sunwapta;1;0;slaveR;;
              sundance;0;0;slaveS;;
```

Figure 5.4 Configuration File for Proposed Implementation

```
FUNCTION m_search(p : position; maxdepth : integer) : integer;
  VAR score, i, value, depth, n_slaves : integer;
      posn : ARRAY[1..MAXWIDTH] of position;
      type pck = record val, num : integer;
                                   { package of info. containing }
                                   { the score and position      }
BEGIN
  posn := generate(p);        { pointer to an array of successors }
  IF empty(posn) THEN
    return(evaluate(p));
    depth = maxdepth;
    score := -pmws(posn[1], depth-1);    { minimal window search }
    FOR (i=1; i<=n_slaves; i++)
      s_send(i, pck);        { send best score and move to slaves }
    WHILE (!empty(posn)) DO BEGIN
                                          { zero window search }
      value := -pvs(p, -score-1, -score, depth-1);
      IF value > score THEN
        value := -pvs(p, -MAXINT, -value, depth-1);
        score := max(score, value);          { critical section }
                                             { to update score  }
    END;
    isdone[0] := 1;                  { master processor is done }
    FOR (i=1; i<=n_slaves; i++)
      IF (!isdone[i])             { wait for all slaves to finish }
        m_process();
  return(score);
END;
```

Figure 5.5 PMWS Algorithm for Root Processor

```
FUNCTION s_search(p : position; maxdepth : integer) : integer;
  VAR score, value, depth : integer;
      posn : ARRAY[1..MAXWIDTH] of position;
      type pck = record val, num : integer;

BEGIN
  posn := generate(p);
  IF empty(posn) THEN
    return(evaluate(p));
  m_receive(pck);                  { receive best score from master }
  WHILE (!empty(posn)) DO BEGIN
    value := -pvs(p, -score-1, -score, depth-1);
    IF value > score THEN
      value := -pvs(p, -MAXINT, -value, depth-1);
    m_send(pck);                  { send score for the move searched }
    m_receive(pck);               { receive next move and best score }
                                  { this is handled by m_process()   }
  END;
  pck.val = IDLE;                     { slave processor is now idle }
  m_send(pck);                    { inform master about idleness    }
                                  { m_process() receives this info. }
END;
```

Figure 5.8 PMWS Algorithm for Slave Processors

```
PROCEDURE m_process;
                        { all 'out of bound' signals (interrupts) }
                        { from slaves are handled by this routine }
  VAR num_sl, n_slaves, i : integer;
      sready : ARRAY[1..n_slaves] of integer;
BEGIN
  num_sl := s_poll(n_slaves, sready);              { poll slaves }
  FOR i := 1 to num_sl DO BEGIN          { for each slave that is }
    slaveid := sready[i];                { waiting for service    }
    IF (!isdone[slaveid]) THEN BEGIN       { if slave is not done }
      s_receive(slaveid, pck);          { receive info. from slave }
      IF (pck.val = IDLE) THEN                  { if slave is idle }
        go_help(slaveid);              { find a busy pr. to help }
      ELSE IF better_score(i) THEN BEGIN          { if slave found }
        update;                                   { a better move  }
        s_send(slaveid, pck);           { send best score to slave }
        record_move;
      END;  {else}
    END;  {if}
  END;  {forloop}
END;  {m_process}
```

Figure 5.7 Interrupt Handler Routine for Root Processor

Thus when a slave becomes idle, the master looks up the structure and determines the busy processor for which it can work. Once the master has determined the one that needs help, it will interrupt it and pass the control of the idle slave to this processor. A routine suitable for doing this assignment is shown in Figure 5.8, where *ass_id* is the identification number of the "assigned master" and *slaveid* is the id of the idle slave.

One can say from Figure 5.3 that if SW on level 2 becomes idle, and SS on level 2 is still working, SW will be assigned to help SS as its slave. All the processes on level 3 are dormant until one processor on level 2 becomes idle and its "assigned master" needs help. Then the slave processes of the "assigned master" become active, but all other level 3 processes remain dormant.

Consider Figure 5.9 to see how the search takes place once the idle processor has been assigned to help a busy one. Assume that the idle processor is SW and it is assigned to help the processor SS. The processor SW is not deployed to help at node *b* (level 2) because from the experimental results presented in Chapter Four, one can infer that on an average only 1.9 moves are searched at cutoff nodes. Thus one would be better off applying the slave to the search of nodes at *level* 3 (like nodes *c* and *d*). At such nodes the search is similar to the search at the root node, with SS as the master and SW as the slave. Processor SS has the capability of working as a slave of itself, thus the subtree search at *level* 3 node is equivalent to that of a two processor case. Once the movelist at node *c* is empty and the slave processor terminates the search, it goes back to node *b* and waits for the master processor to finish. The master does not pass the control of the processor SW back to the root, because this will result in extra communication costs to schedule it again. Also, since this is always a two processor node, the idle time is not significant. For a two processor case, the idle time is a mere 1.0% of the total search

```
PROCEDURE gohelp(slaveid : integer);
       { This routine assigns the idle slave to a busy processor }
  VAR ass_id : integer;
      ass_master : ARRAY[1..slaveid] of integer;
BEGIN
  ass_id := ass_master[slaveid];     { find id of assigned master }
  IF (ass_id = BUSY) THEN            { if assigned master is busy }
    s_send(ass_id, slv_pck);          { send id of idle processor }
END;
```

Figure 5.8 Algorithm for Finding the Assigned Master.

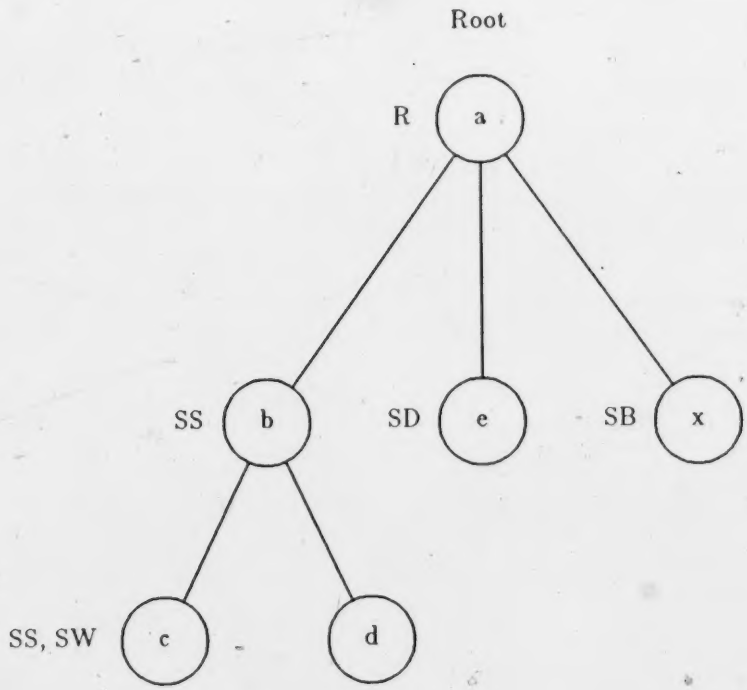


Figure 5.9 Tree Search After Re-scheduling of Processors

time (Figure 4.3). Thus it is worthwhile to let this processor wait at node *b* if the movelist at node *b* is not empty. When processor SS also finishes its move and becomes available, it also goes back to node *b* and gets reassigned with processor SW to the next move in the movelist. Both processors will make the same next move at node *b* to a *level* 3 node (like *d*) and then split the subtree at that node.

Algorithms for doing such a search are shown in Figure 5.10, Figure 5.11 and Figure 5.12. The interrupts to the "assigned master" from either the root process or the new slave process are handled by the *slave_process* routine, Figure 5.10. The functions *ass_mas_search* (Figure 5.11) and *slave_search* (Figure 5.12) illustrate the search of the subtree at *level* 2 nodes by the two processors. The *am_receive* and *am_send* functions are used for communication between the slave processor and its "assigned master".

This scheme reduces processor idleness but raises questions about the cost of doing the extra work in deploying processors. Is there a substantial gain in spite of the extra overhead? To estimate the extra cost associated with the implementation of the scheme outlined above, the following variables are defined:

t_i = time to send message from an idle slave to master.
t_s = time for master to check its structure and determine the processor to be helped.
t_c = time for master to interrupt busy processor and pass control of the idle processor.
t_p = time for passing the path ab or bc to the slave.
t_a = time for slave processor to make the move from a to b.
t_b = time for slave processor to make the move from b to c.

Looking at the processor tree and following the scheme, we can estimate the extra time spend in re-scheduling activity. This is given by the equation:

$$t_{ov} = t_i + t_s + t_c + (t_p + t_a) + (t_p + t_b)$$
$$= t_i + t_s + t_c + t_a + t_b + 2t_p \quad (1)$$

To estimate the gain obtained from employing such a scheme, consider Figure 5.9

```
PROCEDURE slave_process;
                { This routine handles all the interrupts to the }
                { assigned master from root and slave processors }
  VAR slaveid, i : integer;
      type pck = record val, num : integer;

BEGIN
  IF mess_from_root THEN BEGIN        { if interrupt from root pr.}
    m_receive(slv_pck);             { receive id of idle processor }
    s_send(slaveid, path_r_2);        { send path to level 2 node }
  END;
  IF (!isdone[slaveid]) THEN BEGIN              { if slave not done }
    s_receive (slaveid, pck);         { receive info. from slave }
    IF (pck.val = DONE) THEN                    { if slave is done }
      isdone[slaveid] := 1;
    ELSE IF better_score(i) THEN BEGIN
      update;
      s_send(slaveid, pck);           { send best score and move }
      record_move;
    END;
  END;
END;
```

Figure 5.10 Interrupt Handler Routine for Assigned Master

```
FUNCTION ass_mas_search(p : posn; depth : integer) : integer;
          { This routine illustrates the search performed by the }
          { assigned master when there is a slave processor to   }
          { help, it also shows the necessary communication req. }
  VAR posn2, posn3 : posn;
      slaveid, value, score : integer;
      type pck = record val, num : integer;

BEGIN
  FOR (!empty(posn2)) DO BEGIN              { if there is a move }
    IF slave_avail THEN BEGIN     { if a idle slave is available }
      s_send(slaveid, path_2_3);          { send path to slave pr.}
      posn3 := generate(p3);     { generate moves at level 3 node }
      IF empty(posn) THEN
        return(evaluate(p3));
      s_send(slaveid, pck);              { send best score to slave }
      FOR (!empty(posn3)) DO BEGIN
        value := -pvs(p, -score-1, -score, depth-1);
        IF value > score THEN
          value := -pvs(p, -MAXINT, -value, depth-1);
          score := max(score, value);
      END;
      isdone[0] := 1;                    { master processor is done }
    END;
  END;
  return(score);
END;
```

Figure 5.11 Search Algorithm for Assigned Master

```
FUNCTION slave_search(p : posn; depth : integer) : integer;
           { This routine illustrates the search performed by a }
           { a slave processor when helping an assigned master  }
           { and shows the necessary communications required    }
  VAR posn2, posn3 : posn;
      slaveid, value, score : integer;
      type pck = record val, num : integer;

BEGIN
  IF first_ass_mess THEN     { if first message from ass. master }
    am_receive(path_r_2);       { receive path from ass. master }
  FOR (!empty(posn2)) DO BEGIN
    am_receive(path_2_3);        { receive path to level 3 node }
    am_receive(pck);                  { receive the best score }
    FOR (!empty(posn3)) DO BEGIN
      value := -pvs(p, -score-1, -score, depth-1);
      IF value > score THEN
        value := -pvs(p, -MAXINT, -value, depth-1);
      am_send(pck);            { send score found for the search }
      am_receive(pck);        { receive next move and best score }
    END;
    pck.val := DONE;                                { slave is done }
    am_send(pck);                { tell master that it is DONE }
  END;
  return(score);
END;
```

Figure 5.12 Search Algorithm for Slave Processor

again. The processor SW (idle processor) normally helps processor SS with moves at node c. However it is possible that there might be other moves to be searched at node b. Processor SS will need help in searching those moves. If such a situation arises, where more moves need to be searched, processor SS will pass the path of the other *level* 3 nodes (say 'bd') to processor SW so that it can traverse to node d and be able to help in the search of the subtree at that node. This passing of path 'bd' will involve an extra communication of $(t_p + t_b)$. However, since on an average only 1.9 moves are examined at cutoff nodes, the extra cost that may be incurred will not exceed $0.9(t_p + t_b)$, because one move 'bc' has already been examined. Therefore, the total gain in time for this single subtree search is given by,

$$t_g = t_{ps} - (\frac{t_{ps}}{s2} + t_{ov}) - 0.9(t_p + t_b) \qquad (2)$$

where t_{ps} is the time it would have taken for the search of node b by one processor alone and $s2$ is the speedup achieved by using two processors for the search.

Every time an idle processor is assigned to help another busy processor, an overhead of t_{ov} is involved. But every time a processor is helped, its working time can be reduced by about half by spending just t_{ov} extra units of time. The experimental results presented in next section will show that t_{ov} is small in comparison to the savings accrued.

5.3. Experimental Results

There are some unknown constants in the equation for the proposed model that need to be determined to see the effectiveness of the scheme proposed. Among them are the timing variables like t_i, t_c and others. Also, to compare the gains accrued, one needs an estimate of the time spent by processors idling. The timing

variables and the idle times are determined in the next two sub-sections.

5.3.1. Processor Idle Time

To determine the processor idle time, experiments were carried out on the parallel chess program. Each slave processor was made to record the time at the start and finish of each search. Figure 5.13 shows the timers put in the *m_search* (Figure 5.5) and *s_search* (Figure 5.6) routines to get the necessary data for measuring the idle time. This data can be collected from each processor and used to calculate the processor idle time. The experiments were conducted for chess board positions from the Bratko-Kopec's experiment, on a system of four processors. Results from these positions are presented in Table 5.1, some atypical board positions like D, F, G, H and V (Table 4.2) are omitted from the table for calculation of average values.

In a four processor system, for the search of the tree, from 1 to 3 processors become idle. The first three columns on processor idling show the time for which each processor is idle. The fourth columns gives the total idle time for the search of a move (board position). The next two columns have been taken from Table 4.2. It can be seen from Table 5.1 that in general the speedup is more if the idle time is less, this may not always be true because the SO and CO also determine the speedup. Average idle time for a 5-ply search / 4 processor system can be calculated from the average values in Table 5.1

$$\begin{aligned} \textit{Average Idle Time} &= I_1 + I_2 + I_3 \\ &= 118 + 81 + 32 \\ &= 231 \end{aligned}$$

The idle time expressed as a percentage of t_{seq} is given by the equation (7) in chapter four. Substituting the values from Table 5.1, one can get

```
FUNCTION m_search(p : position; maxdepth : integer) : integer;
  VAR score, i, value, depth : integer;
      posn : ARRAY[1..MAXWIDTH] of position;
BEGIN
    .
    M_TIME_S();              {time stamp at start of search}
    score := pmws(posn[1], depth-1);
    .
    .
    isdone[0] := 1;
    M_TIME_E();               {time stamp at end of search}
    .
        m_process();
    M_TIME_PF();      {time stamp when all processor finish}
  return(score);
END;


FUNCTION s_search(p : position; maxdepth : integer) : integer;
  VAR score, i, value, depth : integer;
      posn : ARRAY[1..MAXWIDTH] of position;
BEGIN
    .
    S_TIME_S();              {time stamp at start of search}
    m_receive(pck);
    .
    .
    pck.val = IDLE;
    m_send(pck);
    S_TIME_E();               {time stamp at end of search}
  return(score);
END;
```

Figure 5.13 Processor Idle Time Measurement

Processor Idle Time for a 5-ply Search						
Board No.	I_1 (secs)	I_2 (secs)	I_3 (secs)	I_{idle} (secs)	t_{seq} (secs)	Speedup
B	32	27	9	68	595	3.5
C	108	59	37	204	1057	3.2
E	30	29	16	75	2355	3.7
I	164	159	45	368	1599	3.3
J	223	161	56	440	2243	2.9
K	58	38	13	109	1182	3.5
L	34	25	8	67	749	3.6
M	11	3	4	18	974	3.8
N	38	27	9	74	518	3.0
O	64	48	19	131	460	3.0
P	184	133	46	363	986	2.9
Q	79	59	28	166	1245	3.3
R	4	2	1	7	1765	3.7
S	39	28	7	74	759	3.5
T	193	142	45	380	2299	3.2
U	37	17	6	60	1719	3.6
W	435	270	59	764	1570	3.3
X	50	29	17	96	1341	3.6
Aver.	118	81	32	231	1148	3.4

where,

I_1 : Idle Time for Processor 1

I_2 : Idle Time for Processor 2

I_3 : Idle Time for Processor 3

I_{idle} : Total Idle Time

t_{seq} : Search Time for One Processor Case (Table 4.2)

Speedup : Speedup for the Four Processor Case (Table 4.2)

Table 5.1 Processor Idle Times for a 4 Processor System

$$Sync.\ Loss = \frac{Av.\ t_{idle}}{t_{seq}}$$

$$= \frac{231}{1140}$$

$$= 20\%$$

The next section presents the data for the cost involved in re-scheduling processors. This can then be compared to the processor idle time to determine the gains accrued from employing the scheme.

5.3.2. Processor Re-scheduling Overhead

This section presents some results obtained on the overhead involved in deploying idle processors to assist the busy ones. The main variables involved in "re-scheduling overhead" are:

a) time to check if the processor that the idle processor can help is busy and needs help (t_s).

b) time taken by the master to interrupt the busy processor and pass control of the idle processor (t_c).

c) time taken for passing the search path to the idle processor (t_p).

d) time to traverse the search path passed (t_a, t_b).

e) time to interrupt the busy processor and set it up to accommodate the new helper.

Most of the above activities involve message passing, thus communication overhead will increase in the process of re-scheduling the idle processors. Experiments were done on SUN workstations to find out the above re-scheduling overhead. The variables t_i and t_p can be determined by timing the message transfer between processors, and the timing of the routine *gohelp* (Figure 5.8) gives the value of the

variables t_s and t_c. The results are as follows:

t_i = time to send message from an idle slave to master.
= 9.1 msec
t_s = time for master to check its structure and determine the processor to help.
= 0.1 msec
t_c = time for master to interrupt busy processor and pass control of the idle processor.
= 11.2 msec
t_p = time for passing the path ab or bc to the slave.
= 10 msec
t_a = time for slave processor to make the move from a to b or b to c.
= 0.6 msec

It can be observed from the results that the maximum time is spent in inter-processor communication to pass control and moves to processors. These results on processor idle time and processor re-scheduling overhead are analyzed with respect to the gains accrued by such a scheme in the section below.

5.4. Conclusion on the Effectiveness of the Proposed Scheme

The previous sections have presented a detailed analysis of the synchronization problem. From the experimental results and equation (1), one can calculate the extra cost of re-scheduling the processors. The extra cost is given by the equation

$$t_{ov} = t_i + t_s + t_c + 2t_a + 2t_p$$

Substituting the values obtained from the experimental results we get,

$$t_{ov} = 9.1 + 0.1 + 11.2 + 2\times0.6 + 2\times10$$
$$= 42 msec.$$

This is a small quantity when compared to the idle times of the processors (Table 5.1). From 1 to (N-1) processors become idle, and the results for the four processor system (Table 5.1) show that on an average 1 to 3 processors are idle for 118 secs., 81 secs. and 32 secs. respectively. From equations (1), (2) and table 5.1, one can estimate the idle time for such an implementation.

$$Sync.Loss = \frac{Av.\ t_{idle}}{t_{seq}}$$

$$= \frac{135}{1140}$$

$$= 12\%$$

One can see that the proposed scheme reduces the idle time from 20% to 12%, this is a 40% reduction in idle time of processors. This reduction is at the expense of an increased commmunication overhead, which is necessary to re-schedule the idle processors to help the busy ones. This results in an increase of communication overhead from 3% to about 4%. This is less when compared to 8% reduction in idle time. This clearly shows that a large gain is possible by deploying idle processors to help busy ones.

This scheme rests heavily on efficient interprocessor communication and processor scheduling. With even faster machines, efficient interrupt handling and fast communication networks, this scheme becomes even more attractive. The results obtained form the necessary base to develop guidelines for an implementation suitable to a particular software and system configuration.

Chapter 6

CONCLUSIONS

This thesis has discussed problems with the parallel implementation of the chess program that uses the parallel alpha-beta search algorithm with appropriate enhancements. The experimental results for Marsland's analytical model showed that the search strategy (PVS) used in the parallel version of Parabelle [MARS85], implemented on a network of SUN workstation, is efficient in causing cutoffs in tree searches; on an average only 1.9 moves are examined at cutoff nodes. This, coupled with local transposition and other enhancements, yields good speedups for 4 processors.

If more processors are used, the overhead increases. Eventually a point will be reached beyond which the addition of an extra processor will yield no benefit. This dead end can only be avoided by minimizing the overhead. Communication and search overheads seem to be within acceptable limits. It is the synchronization loss that is responsible for causing this dead end. The synchronization problem has been analyzed in detail and a scheme has been proposed to reduce the synchronization losses. The proposed scheme attempts to deploy idle processors to help the busy ones and hence reduce the idle time. The experimental results show that the gains obtained from such a scheme far outweigh the extra cost involved, and thus reduce the synchronization losses considerably. This solution is suitable for at least a small number of processors.

At the time of this thesis work, there were no experimental results for parallel alpha-beta implementation beyond eight processors [NEWB82]. Virtual Tree Machine [BURT84] simulation results predict good speedups, but because of certain inherent properties of the alpha-beta search tree no system has actually been able

to give even near the predicted speedup. Typically, the theoretical models and simulation results do not properly account for the real costs that are the inevitable price an implementation must pay. More research needs to be done on the actual implementation of such algorithms on a large processor system to be able to resolve issues associated with the same and come up with a "general" efficient parallel solution.

6.1. Directions for Future Work

The process of designing efficient parallel programs for multi-processor systems, especially for computer chess, is still in its experimental phase. The work done to date has identified many problems associated with parallel implementation. There are certain ideas that look promising in increasing the speedup of alpha-beta search. Even more cutoffs could be obtained if the new alpha bound were made available to all the processors as soon as it was determined, rather than waiting for the slave processor to finish its subtree search. A further reduction in processor idle time and hence a corresponding improvement in speedup could be obtained if more than one processor is allocated for searching new principal variations. This is because the better cutoff value associated with the new variation will be used earlier in the search of all the subsequent moves in the movelist. For search of small trees, a strong ordering of moves is more important. Reduction in idle time and hence better speedups can be achieved if subtrees that require more time to search are scheduled earlier in the tree search. This can be achieved only if one can determine the nature of subtree searches before scheduling them to respective slave processors.

All implementations of game tree search algorithms have been on systems running under a uniprocessor operating system. ZMOB is an attempt towards a truly distributed operating system [BANE81a]. Design of algorithms for such a

system will require a different approach. For implementation on a VTM, the user determines the scheduling of work; but under a distributed operating system, the parallel execution will be determined by the system. It has to be seen how such a system can be effectively used for parallel search of game trees.

With all these promising ideas, one can expect to obtain a satisfactory speedup, but still it is another matter to come up with an implementation whose speedup increases asymptotically with the number of processors employed. Most of the parallel programs are a result of the parallelization of sequential programs, and thus encounter many problems in their implementation. Until a different approach is taken and parallel programming is better understood, asymptotic behavior will remain a distant goal.

References

AKL82 S. Akl et. al., "Design, Analysis, and Implementation of a Parallel Tree Search Algorithm", *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-4*, 2, pp. 192-203, 1982.

BAER76 J.L. Baer, "Multiprocessing Systems", *IEEE Trans. on Computers*, December 1976.

BANE81 B. Bane et. al., "ZMOB: A New Computing Engine for AI", *Proc. IJCAI-81, Vancouver*, pp 955-960, Aug. 1981.

BANE81a B. Bane et. al., "Operating System for ZMOB", *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, pp 125-132, Nov. 1981.

BAUD78 G.M. Baudet, "The Design and Analysis of Algorithms for Asynchronous Multiprocessors", *Ph.D. Thesis, Dept. of Computer Science, CMU*, April 1978.

BRAT82 I. Bratko and D. Kopec, "A Test for Comparision of Human and Computer Performance in Chess", *Advances in Computer Chess 3*, M.R.B.Clarke (ed.), Pergamon Press, pp 31-56, 1982.

BURT84 F.W. Burton and M. Huntbach, "Virtual Tree Machines", *IEEE Transactions on Computers C-33, 3*, pp. 278-280, March 1984.

CAMP83 M. Campbell and T.A. Marsland, "A Comparision of Minimax Tree Search Algorithms", *Artificial Intelligence 20*, pp 347-367, 1983.

CORA76 L. Coraor and J. Robinson, "Using Parallel Microprocessor in Tree Decision Problems", *Proceedings of the International Symposium on Mini and Micro Computers, IEEE*, pp 51-55, 1976.

ENSL77 P.H. Enslow, "Multiprocessor Organization - A Survey" *Computing Surveys*, Vol. 9, No. 1, March 1977.

FISH80 J. Fishburn and R. Finkel, "Parallel alpha-beta Search on Arachne", *TR 394, Computer Science Dept., Univ. of Wisconsin, Madison*, 1980

FISH81 J.P. Fishburn, "Analysis of Speedup in Distributed Algorithms", *Ph.D. Thesis, Dept. of Computer Science, Univ. of Wisconsin*, 1981.

FLYN66 M.J. Flynn, "Very High Speed Computing Systems", *Proc. IEEE*, Vol. 54, December 1966.

GILL78 J. Gillogly, "Performance Analysis of the Technology Chess Program, *Ph.D. dissertation, Computer Science Dept., CMU, Pittsburg,* March 1978.

JONE80 A.K. Jones and P. Schwarz, "Experience using Multi-processor System - A Status Report", *Computing Surveys,* Vol. 12, No. 2, June 1980.

KNUT75 D. Knuth and R. Moore, "Analysis of the Alpha-beta Pruning Algorithm", *Artificial Intelligence 6,* pp. 293-326, 1975.

KUCK68 D.J. Kuck et. al., "The ILLIAC IV Computer", *IEEE Trans. Computer,* C-17, 1968.

LEFF83 S.J. Leffler et. al., "A 4.2BSD Interprocess Communication Primer (Draft)", *Computer System Research Group, Univ. of California, Berkeley,* December 1983.

MARS81a T.A. Marsland and M. Campbell, "A Survey of Enhancements to the Alpha-beta Algorithm", *ACM81 Conference Proceedings, Los Angeles,* 109-114, 1981.

MARS82 T.A. Marsland and M. Campbell, "Parallel Search of Strongly Ordered Game Trees", *Computing Surveys,* 14, 4, pp. 533-551, 1982.

MARS85 T.A. Marsland and F. Popowich, "Parallel Game-tree Search", *IEEE Transaction on PAMI,* (to appear) May 1985.

MYRI84 *Myrias 4000 System Description,* Myrias Research Corporation, Edmonton, May 1984.

NEWB82 M. Newborn, "OSTRICH/P - A Parallel Search Chess Program", *Technical Report SOCS - 82.8, School of Computer Science, McGill Univ., Montreal,* March 1982.

OLAF85 M. Olafsson and T. Marsland, "Implementation of Virtual Tree Machines", *Technical Report 85-9, Computing Science Dept., Univ. of Alberta, Edmonton,* May 1985.

POPO83 F. Popowich and T.A. Marsland, "Parabelle: Experience with a Parallel Chess Program", *TR 83-7, Computing Science Dept., Univ. of Alberta, Edmonton,* August 1983.

RIEG79 C. Rieger, "ZMOB : A Mob of 256 Cooperative Z-80A based Microcomputers", *Proceedings of the ARPA Vision Workshop,* LosAngeles, CA, 1979.

SCHA83 J. Schaeffer, "The History Heuristic", *ICCA Journal 6(3)*, pp. 16-19, 1983.

SCHA84 J. Schaeffer, M. Olafsson and T.A. Marsland, "Experiments in Distributed Tree Search", *TR 84-4, Computing Science Dept., Univ. of Alberta, Edmonton*, June 1984.

SLAG69 J.R. Slagle and J.K. Dixon, "Experiments with some Programs that Search Game Trees", *Journal ACM*, pp. 198-207, 1969.

SWAN77 R. Swan et al., "Cm* : A Modular Multi-microprocessor", *AFIPS 1977 National Comp. Conf.*, June 1977.

THOM82 K. Thompson and J.H. Condon, "Belle Chess Hardware", *Advances in Computer Chess 3, Pergammon Press*, 1982.

TRIG80 R. Trigg et. al., "ZMOB : A Highly Parallel Multiprocessor", *Technical Report 911, Dept. of Computer Science, Univ. of Maryland*, May 1980.

WULF72 W. Wulf and C. Bell, "C.mmp - A Multi-miniprocessor", *Fall Joint Computer Conf.*, December 1972.

Supplementary Bibliography

BENT79 J.L. Bentley and H.T. Kung, "Two Papers on a Tree Structured Parallel Computer", *Technical Report, Dept. of Computer Science, CMU*, August 1979.

BHAN77 D.P. Bhandarkar, "Some Performance Issues in Multiprocessor System Design", *IEEE Transactions on Computers*, pp 506-511, May 1977.

BRIG81 F. Briggs and M. Dubois, "Efficient Interprocess Communications for MIMD Multiprocessor Systems", *The 8th Annual Symposium on Computer Architecture*, May 1981.

BRIG82 M. Dubois and F.A. Briggs, "An Approximate Analytical Model for Asynchronous Processes in Multiprocessors", *Proc. of the 1982 International Conf. on Parallel Processing*, Aug. 1982.

ENSL74 Comtre Corporation and P.H.Enslow (Ed), "Multiprocessors and Parallel Processing", *John-Wiley and Sons*, NY, 1974.

FINK82 R. Finkel and J. Fishburn, "Parallelism in Alpha-Beta Search", *Artificial Intelligence*, pp. 89-106, 1982.

FISH79 R.A. Fishburn et. al., "Two papers on Alpha-Beta search", *Technical Report, Dept. of Computer Science, Univ. of Wisconsin*, 1979.

HARR77 J.A. Harris and D.R. Smith, "Hierarchical Multiprocessor Organizations", *Proceedings of the 4th Annual Symposium on Computer Architecture, Silver Springs, MD*, March 1977.

HELL78 D. Heller, "A Survey of Parallel Algorithms in Numerical Linear Algebra", *SIAM Review*, pp. 740-777, October 1978.

KAIN75 R.Y. Kain and K.V. Sastry, "On the Performance of certain Multiprocessor Computer Organization", *IEEE Trans. on computers*, November 1975.

KART81 S.I. Kartashev and S.P. Kartashev, "Reconfiguration of Dynamic Architecture into Multicomputer Network", *Proceedings of the 1981 International Conference on Parallel Processing*, 1981.

KUCK76 D.J. Kuck, "Parallel Processing of Ordinary Programs", *Advances in Computers, Vol. 15*, Academic Press, 1976.

KUNG80 H.T. Kung, "The Structure of Parallel Algorithms", *Advances in Computer, Vol. 19*, Academic Press, 1980.

LIND83 G. Lindstrom, "The Key Node Method: A Highly-Parallel Alpha-Beta Algorithm", *Technical Report UUCS 83-101, Dept. of Computer Science, Univ. of Utah, Salt Lake City*, March 1983.

MACK84 B. Mackinlay, "The SUN-2 Workstation", UNIX/WORLD, *A Tech Valley Publication*, Vol.1, No.5, Oct. 1984.

MARS83 T.A. Marsland,"Relative Efficiency of Alpha-Beta Implementations", *IJCAI Proceedings, Karlsruhe*, August 1983, 763-766.

OLEN78 P.N. Olenick and S.H. Fuller, "The Implementation and Evaluation of a Parallel Algorithm on C.MMP", *Technical Report, Dept. of Computer Science, CMU*, June 1978.

PAGE76 R.G. Arnold and E.W. Page, "A Hierarchical Re-structurable Multi Microprocessor Architecture", *Proceedings of the 3rd Annual Symposium on Computer Architecture, Clearwater, FL*, pp. 40-45, January 1976.

PEAR80 J. Pearl, "Asymptotic Properties of Minimax Trees and Game Searching Procedures", *Artificial Intelligence 14*, pp 113-138, 1980.

RIEG81 C. Rieger, "ZMOB : Hardware from a User's Viewpoint", *Proc. IEEE Workshop Pattern Recog. and Image Processing (PRIP)*, Aug. 1981. (also UOM CS TR-1042)

RIEG81a C. Rieger, "ZMOB : Doing it in Parallel !", *Proc. IEEE Workshop CA-PAIDM*, Nov. 1981.

SATY80 M. Satyanarayanan, "Multiprocessors: A Comparative Study", *Published by Prentice-Hall, Inc.*, 1980.

SHIN81 J. Sasidhar and K.G. Shin, "Design of a General-Purpose Multiprocessor with Hierarchical Structure", *Proceedings of the 1981 International Conference on Parallel Processing*, 1981.

SIEG82 H.J. Siegel et. al., "Design and Simulation of a MC68000 based Multi-microprocessor System", *Proceedings of the 1982 International Conference on Parallel Processing*, 1982.

SLOT62 D.L. Slotnick et. al., "The SOLOMON Computer", *AFIPS Conference Proc.*, 1962.

TRUJ82 V.A. Trujillo, "System Architecture of a Reconfigurable Multi-microprocessor research system", *Proceedings of the 1982*

International Conference on Parallel Processing, 1982.

VALI75 L.G. Valiant, "The Intrinsic Complexity of Parallelism in Comparison Problems", *SIAM J. Computing*, September 1975.

YAU77 S.S. Yau and H.S. Fung, "Associative Processor Architecture - a Survey", *Computing Surveys*, **9**, 1977.